510.84
Il6r
no. 997
cop. 2

Report No. UIUCDCS-R-79-997 UILU-ENG 79 1744

PARALLEL ALGORITHMS FOR NETWORK ROUTING PROBLEMS AND RECURRENCES

by

[redacted]

John A. Wisniewski and Ahmed H. Sameh

November 1979 NSF-OCA-MCS75-21758-000045

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

[illegible]

Report No. UIUCDCS-R-79-997

PARALLEL ALGORITHMS FOR NETWORK ROUTING PROBLEMS AND RECURRENCES[*]

by

John A. Wisniewski[**] and Ahmed H. Sameh[**]

November 1979

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

[*]This work was supported in part by the National Science Foundation under Grant No. US NSF MCS75-21758.

[**]Department of Computer Science, University of Illinois - Urbana, IL 61801.

Abstract

In this paper, we consider the parallel solution of recurrences, and linear systems in the regular algebra of Carré. These problems are equivalent to solving the shortest path problem in graph theory, and they also arise in the analysis of Fortran programs. Our methods for solving linear systems in the regular algebra are analogs of well-known methods for solving systems of linear algebraic equations. A parallel version of Dijkstra's method, which has no linear algebraic analog, is presented. Considerations for choosing an algorithm when the problem is large and sparse are also discussed.

1. Introduction

A basic problem in applications of graph theory is that of finding shortest paths in a weighted graph. This problem arises in finding the shortest or least cost path in transportation problems, in solving minimum cost flow problems, in finding the critical (i.e., longest) path in scheduling problems, and in finding adversary routes through nuclear fuel-cycle facilities [13, 23].

While there are several versions of the shortest path problem [8], we will deal with the problem of finding the shortest paths in definite graphs G from one node to all others, where the path length used is the sum of the weights of the arcs in the path. A digraph G(V,A) is a set V of nodes and a set A of arcs which are ordered pairs of nodes. Assume that the nodes are numbered from 1 to n so that $V = \{1, 2, \ldots, n\}$. Each arc $(i,j) \in A$ has an associated length, $\delta_{i,j}$. By defining $\delta_{i,i} = \infty$, $1 \leq i \leq n$ and $\delta_{i,j} = \infty$ for $(i,j) \notin A$, we have the n by n distance matrix $D = [\delta_{i,j}]$. If a graph G contains no cycles the sum of whose arc weights are less than or equal to zero, then we say that G is definite.

Although extensive work has been done on sequential algorithms for solving the shortest path problem, little work has been done [4, 6, 19] on parallel algorithms for this problem. Levitt and Kautz [19] discuss an iterative algorithm due to Hu [12] which has been tailored to a cellular array machine. Chen and Feng [4] discuss a version of Ford's algorithm [9, 10] for an associative processor machine. Dekel and Sahni [6] present a method for cube connected and perfect shuffle computers. Here, however, we treat the shortest path problem in a more general way, using the regular algebra of Carré [2]. Furthermore, we will assume that:

(i) any number of processors may be used at any time, but we will give bounds on this number,

(ii) each processor may perform either a comparison or any of the four arithmetic operations in one time step, and

(iii) there are no memory or data alignment time penalties.

If p processors are being used, then we denote the computation time as T_p unit steps. Thus T_1 is the time required by a serial machine. We define the speedup of a computation using p processors over the serial computation time as $Sp=T_1/T_p \leq 1$.

Carré [2], has shown that the shortest path problem, as well as many other problems, can be posed as linear systems of the form $x=Ax + b$ in a regular algebra. He gives the examples of the scheduling algebra of Cruon and Hervé [5], the two elements boolean algebra, and a stochastic communication problem (Kalaba [16]; Moisil [20]). Triangular linear systems, or recurrences, in a regular algebra also arise in the analysis of Fortran programs [17]. Consequently, it is important to solve efficiently such recurrences on a parallel computer.

In addition to considering the solution of linear systems in a regular algebra, we will present a parallel version of Dijkstra's algorithm [7] which is applicable only to graphs with positive arc weights. Furthermore, considerations for choosing an algorithm when the graph is sparse, i.e.,has a large number of infinite entries in the distance matrix, are discussed.

1.1. Notation

We follow the convention that capital letters denote matrices, lower case letters denote vectors, and lower case greek letters denote scalars. Except in the cases where we state time and processor bounds, the symbols + and × are taken to represent the generalized addition and generalized multiplication operations of the regular algebra. In the statement of the time and processor bounds, the symbols + and × will take on their normal meaning. We will frequently omit the symbol × when it is clear that a generalized product is to be taken. We will also use the Σ and Π notations for the generalized sum or product of a set of items. Unless otherwise stated, $\log n \equiv \lceil \log_2 n \rceil$ for any positive number n.

1.2. The Regular Algebra

For convenience, we restate the algebra of Carré. We start by defining a semiring $(S,+,\times)$ which satisfies the following properties:

(i)	commutativity	$\alpha + \beta = \beta + \alpha$ $\alpha \times \beta = \beta \times \alpha$
(ii)	associativity	$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$ $\alpha \times (\beta \times \gamma) = (\alpha \times \beta) \times \gamma$
(iii)	distributivity	$\alpha \times (\beta + \gamma) = (\alpha \times \beta) + (\alpha \times \gamma)$
(iv)	idempotency	$\alpha + \alpha = \alpha$

for α, β, γ, ε, S.

The set S has a unit element ε such that

$$\alpha \times \varepsilon = \alpha,$$

and a null element θ satisfying

$$\alpha + \theta = \alpha, \alpha \times \theta = \theta,$$

for all $\alpha \varepsilon S$. Furthermore, we have the law of multiplicative cancellation:

(v) if $\alpha \neq \theta$ and $\alpha \times \beta = \alpha \times \gamma$ then $\beta = \gamma$.

We define for the semiring $(S,+,\times)$ the order relation $\leq$:

$$\alpha \leq \beta \text{ if and only if } \alpha + \beta = \alpha.$$

1.3. Extensions to Matrix Operations.

We now extend the definitions of the regular algebra to matrices all of whose elements belong to the set S. The definitions of matrix addition, matrix multiplication and matrix transposition are analogous to those in linear algebra. Note that matrix addition is idempotent,

$$A + A = A.$$

We define a square m by m unit matrix $I = [\varepsilon_{i,j}]$ with $\varepsilon_{i,j} = \varepsilon$ if $i = j$ and $\varepsilon_{i,j} = \theta$ if $i \neq j$. Note that $IA = AI = I$ for any m by m matrix A. The null matrix N, all of whose elements are θ, satisfies

$$A + N = A$$

$$A \times N = N.$$

The partial ordering for matrices is easily extended:

$$A \leq B \text{ if and only if } \alpha_{i,j} \leq \beta_{i,j} \text{ for all } i,j.$$

In particular, it follows that

$$A \leq B \text{ if and only if } A + B = A.$$

The shortest path problem can now be stated as a linear system in the regular algebra. Let G (V, A) be a graph with distance matrix D. Let the vector b represent an initial labeling of the nodes V (for the vector b, β_i is the inital label on node i). The shortest path problem is equivalent to solving the linear system x = Dx + b in the regular algebra [2]. For the single source problem the vector b is given by by $\beta_s = \varepsilon$ where s is the source node, and $\beta_i = \theta$ otherwise.

2. Solution of Recurrences

In this section we study the solution of systems of the form

$$x = Lx + f, \tag{1}$$

where the matrix L is a strictly lower triangular matrix. These systems arise in the analysis of Fortran programs, and are also used in methods for solving general systems

$$x = Ax + b \ .$$

2.1. The Column Sweep Algorithm

The most fundamental of all algorithms for solving recurrences in the regular algebra is the column sweep algorithm. The solution of x = Lx + f , where

$$L = \begin{bmatrix} \theta & \theta & \cdots & \theta \\ \lambda_{2,1} & \theta & \cdots & \theta \\ \lambda_{3,1} & \lambda_{3,2} & \theta & \theta \\ - & - & - & - \\ \lambda_{n,1} & \lambda_{n,2} & \lambda_{n,n-1} & \theta \end{bmatrix} ,$$

$$x = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix} , \qquad f = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_n \end{bmatrix} , \quad (2)$$

is given by

$$\xi_1 = \phi_1 ,$$

$$\xi_j = \sum_{i=1}^{j-1} \lambda_{j,i} \xi_i + \phi_j , \quad j = 2,3,\ldots,n \qquad (3)$$

Rewriting the system (3) as column operations we get the

<u>Column Sweep Algorithm</u>:

```
1.  Set $x \leftarrow f$
2.  For $i = 1, 2, \ldots, n-1$
      $x \leftarrow x + \xi_i \ell_i$        (in parallel)
```

In the algorithm, ℓ_i represents the ith column of the matrix L. A sequential algorithm for solving recurrence (1) requires $n^2 - n$ operations, i.e. $T_1 = n^2 - n$. On the other hand, the column sweep algorithm can be performed in $2(n-1)$ time steps using at most $n-1$ processors. This gives a speedup of $\frac{n}{2} + O(1)$ over the sequential algorithm with a corresponding efficiency of roughly 1/2.

2.2 The Product Form of the Solution

In this section, the solution x of (1) is formed as a sum of products. Each product involves a power of the matrix L multiplied by the vector f.

Lemma 1. Let L be strictly lower triangular, i.e. L has the form given in equation (2). Also, let

$$L^i = \begin{bmatrix} N & N \\ & \\ R^T & N \end{bmatrix},$$

where R is upper triangular of order n-i, i.e. L^i is lower triangular with i null diagonals in its lower half.

Proof: From the definition of matrix multiplication and the facts that $\theta + \theta = \theta$, $\theta \times \alpha = \theta$ for all $\alpha \in S$, and that L is null on the main diagonal, each multiplication of L^{i-1} by L introduces another diagonal of null elements into the product. ■

Corollary 1: $L^n = N$. ■

Lemma 2: The product $L^i L^j$ $i \leq j$, $i + j < n$ can be formed in $\log(n-i-j) + 1$ steps using at most $p(n-i-j)$ processors, where

$$p(k) = \sum_{i=1}^{k} \sum_{j=1}^{i} j = k(k+1)(k+2)/6 .$$

Proof: This result also follows from the definition of matrix multiplication. Since L^i has i null diagonals and L^j has j null diagonals, the longest dot product formed is of length n-i-j. This establishes the time bound. The processor bound is established by counting the number of multiplications needed to compute the product. This is the stated processor count since all of the multiplications can be formed simultaneously.

■

The product form of the solution is found by substituting for x in the right hand side of (1), its equivalent, $Lx + f$. Thus,

$$x = L^2x + Lf + f.$$

After n-1 such substitutions, we obtain

$$x = L^nx + L^{n-1}f + \ldots + Lf + f.$$

From Corollary 1, $L^n = N$ so

$$x = L^{n-1}f + L^{n-2}f + \ldots Lf + f. \qquad (4)$$

Furthermore, equation (4) can be written in the form

$$x = (I + L^{n/2})(I + L^{n/4}) \ldots (I + L^2)(I + L)\ f. \qquad (5)$$

Heller's algorithm [11] for solving triangular systems of linear algebraic equations, can now be applied.

Algorithm 1. Product form of solution for recurrences.

```
Set $x_0 \leftarrow f$
For $i = 0, 1, \ldots, \nu = \log(n/4)$
    $x_{i+1} \leftarrow (I + L^{2^i})\ x_i$
    $L^{2^{i+1}} \leftarrow L^{2^i}\ L^{2^i}$
$x \leftarrow (I + L^{\frac{n}{2}})\ x_{\nu+1}$
```

Theorem 1: Algorithm 1 can be performed in parallel in $\log^2 n + 3 \log n + 1$ steps using at most $\frac{1}{6} n^3$ processors.

Proof: Heller has already shown that Algorithm 1 can be performed in $\log^2 n + 3 \log n + 1$ steps. The formation of x_{i+1} and x requires only $O(n^2)$ processors whereas the formation of $L^{2^{i+1}}$ requires $O(n^3)$ processors. In particular, the largest number of processors are required in forming L^2. From Lemma 2, this number is

$$p(n-2) = \frac{1}{6}(n-2)(n-1)n < \frac{1}{6}n^3$$

processors. ■

2.3 Product of Elementary Matrices Solution

The method presented in this section is an analog of Sameh and Brent's Algorithm I [22]. It is superior to Algorithm 1 above since it takes approximately half the time while using significantly fewer processors to solve the recurrence (1).

<u>Definition:</u> M_j is an elementary matrix, if it is of the form,

$$M_j = \begin{bmatrix} \theta & & & & & & & \\ & \theta & & & & N & & \\ & & \ddots & & & & & \\ & & & \theta & & & & \\ N & & & \mu_{j+1,j} & \theta & & & \\ & & & \mu_{j+2,j} & & \ddots & & \\ & & & \vdots & N & & \ddots & \\ & & & \mu_{n,j} & & & & \theta \end{bmatrix},$$

M_j has all null entries except in column j below the main diagonal. ■

<u>Lemma 3:</u> Let M_i and M_j be elementary matrices, then $M_iM_j = N$ if $i \leq j$. Furthermore if $i > j$ then $P = M_iM_j$ has all null elements except in column j below the ith row.

Proof: M_j has its first j rows all null and M_i has all null elements except in column i below the main diagonal. Thus if $i \leq j$ all inner products of rows of M_i with columns of M_j must be null, i.e. $M_i M_j = N$. If $i > j$ then the only non-null inner products of $P = M_i M_j$ occur when the inner products of rows i+1, i+2, ..., n of M_i are taken with column j of M_j. ■

Lemma 4: Let $M_1, M_2, \ldots, M_j, M_{j+1}$ be elementary matrices. Then the product

$$(I + M_{j+1})(I + M_j) \ldots (I + M_1)(I + M_1)(I + M_2) \ldots (I + M_{j+1}) =$$

$$(I + M_{j+1})(I + M_j) \ldots (I + M_1). \qquad (6)$$

The proof is by induction on j.

Basis step: For j = 1, we have

$$(I + M_1)(I + M_1) = I + M_1 + M_1 + M_1^2.$$

From Lemma 3 and idempotency

$$I + M_1 + M_1 + M_1^2 = I + M_1 + M_1 = I + M_1,$$

establishing the basis.

Induction step: Assume that the assertion holds for index j, and let

$$\overline{M}_j = (I + M_j)(I + M_{j-1}) \ldots (I + M_2)(I + M_1).$$

From the inductive hypothesis we have

$$(I + M_{j+1})(I + M_j) \ldots (I + M_1)(I + M_1)(I + M_2) \ldots (I + M_{j+1}) =$$

$$(I + M_{j+1}) \overline{M}_j (I + M_{j+1}).$$

Furthermore,

$$\overline{M}_j(I + M_{j+1}) = \overline{M}_j + \overline{M}_jM_{j+1}. \tag{7}$$

By repeated applications of Lemma 3 we obtain

$$\overline{M}_jM_{j+1} = M_{j+1} ,$$

hence (7) becomes

$$\overline{M}_j(I + M_{j+1}) = \overline{M}_j + M_{j+1}.$$

Consequently,

$$\begin{aligned}(I + M_{j+1})\overline{M}_j(I + M_{j+1}) &= (I + M_{j+1})(\overline{M}_j + M_{j+1}) \\ &= \overline{M}_j + M_{j+1}\overline{M}_j + M_{j+1} + M_{j+1}{}^2 \\ &= \overline{M}_j + M_{j+1}\overline{M}_j + M_{j+1} \\ &= \overline{M}_j + M_{j+1}(\overline{M}_j + I).\end{aligned}$$

Since I is an element of the expanded sum of $\overline{M}_j$, then due to idempotency, we have

$$\begin{aligned}(I + M_{j+1})\overline{M}_j(I + M_{j+1}) &= \overline{M}_j + M_{j+1}\overline{M}_j \\ &= (I + M_{j+1})\overline{M}_j,\end{aligned}$$

establishing the lemma. ■

In a similar fashion, it can be shown that

$$(I + M_1)(I + M_2) \dots (I + M_{j+1})(I + M_{j+1})(I + M_j) \dots (I + M_1) =$$
$$(I + M_{j+1})(I + M_j) \dots (I + M_1). \tag{8}$$

Let

$$(I + L)^* \equiv (I + M_{n-1})(I + M_{n-2}) \dots (I + M_2)(I + M_1) . \tag{9}$$

Since

$$(I + L) = (I + M_1)(I + M_2) \ \ldots \ (I + M_{n-2})(I + M_{n-1}) ,$$

from Lemma 4 and equation (8), we see that

$$(I + L)^*(I + L) = (I + L)^* = (I + L)(I + L)^* . \qquad (10)$$

We now introduce a lemma of fundamental importance to what follows.

Lemma 5: The vector x solves the system $x = Ax + f$ if and only if x solves the system $x = x + Ax + f$.

Proof: If $x = Ax + f$ then adding x to both sides and using idempotency gives

$$x + x = x + Ax + f,$$

$$x = x + Ax + f.$$

Hence,

$$x + Ax + f \leq Ax + f,$$

i.e., either $x = x$ or $x = Ax + f$. Since $x = x$ is redundant, $x = Ax + f$. ■

The following result was originally given by Carré [2], but we state and prove it in terms of elementary matrices.

Theorem 2: Let $x = (I + L)^* f$, then x satisfies $x = Lx + f$.

Proof: From Lemma 5, $x = Lx + f$ is equivalent to $x = (I + L)x + f$, or

$$x = (I + L)(I + L)^* f + f,$$

$$= (I + L)^* f + f.$$

From equation (9) f is an element of the expanded sum of $(I + L)^* f$, thus $x = (I + L)^* f$ satisfies (11). ■

From Theorem 2, we can write the solution x to equation (1) as

$$x = (I + L)^* f = (I + M_{n-1})(I + M_{n-2}) \dots (I + M_2)(I + M_1)f,$$

which motivates the following algorithm.

<u>Algorithm 2</u>: Solution of recurrences in product form.

```
1. Set $(I + M_i)^{(0)} = I + M_i$, $i=1, \dots, n-1$; $f^{(0)} = f$.

2. For $j=0, 1, \dots, \nu = \log(n/4)$
   | Form $(I + M_i)^{(j+1)} = (I + M_{2i+1})^{(j)}(I + M_{2i})^{(j)}$
   |         in parallel for $i=1, 2, \dots, n/2^{j+1} - 1$ .
   | Form $f^{(j+1)} = (I + M_1)^{(j)} f^{(j)}$
   |_

3. Set $x = (I + M_1)^{(\nu)} f^{(\nu)}$.
```

Algorithm 2 is a direct analog of Algorithm I of Sameh and Brent [22] for triangular linear algebraic equations. Hence, their results (Theorem 2.1) apply. The solution x to the recurrence (1) can be found in

$$\tau = \frac{1}{2}\log^2 n + \frac{3}{2}\log n + 3$$

steps using no more than

$$\pi = (15/1024)n^3 + 0(n^2) = n^3/68 + 0(n^2)$$

processors.

2.4 Limited Parallelism

In this section we present two methods which are analogs of those of Hyafil and Kung [15] for solving the recurrence (1). These methods are used when the number of processors p is fixed.

The first method utilizes the algorithm decomposition applied to the column sweep algorithm:

$$x^{(1)} = f$$
$$x^{(i+1)} = (I + M_i)x^{(i)}, \; i=1,\, 2,\, \ldots,\, n-1.$$

It is easy to see that

$$(I + M_i)x^{(i)} = \begin{bmatrix} \xi_1^{(i)} \\ \cdot \\ \cdot \\ \cdot \\ \xi_i^{(i)} \\ \xi_{i+1}^{(i)} + \lambda_{i+1,i} \quad \xi_i^{(i)} \\ \cdot \\ \cdot \\ \cdot \\ \xi_n^{(i)} + \lambda_{n,i} \quad \xi_i^{(i)} \end{bmatrix}$$

Thus, given p processors, the product $(I + M_i)x^{(i)}$ can be formed in $2\left\lceil \frac{n-i}{p} \right\rceil$ steps. Since $T_p(n) = \sum_{i=1}^{n} 2\left\lceil \frac{n-i}{p} \right\rceil$ we have from Hyafil and Kung [15] that

$$T_p(n) \le \frac{n^2}{p} + (2 - \frac{3}{p})n + \frac{2}{p} - 2.$$

This method is most practical when $p < n$.

The second method uses the problem decomposition principle.

Let

$$L = \begin{bmatrix} L_{1,1} & & & \\ L_{2,1} & L_{2,2} & & N \\ \vdots & \vdots & \ddots & \\ L_{m,1} & L_{m,2} & \cdots & L_{m,m} \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix},$$

then equation (1) can be written as,

$$\begin{aligned} x_1 &= L_{1,1}x_1 + f_1, \\ x_2 &= L_{2,1}x_1 + L_{2,2}x_2 + f_2, \\ &\vdots \\ x_m &= (\sum_{j=1}^{m} L_{m,j}x_j) + f_m. \end{aligned}$$

This leads to the following algorithm.

Algorithm 3: Recurrence solver with limited number of processors.

```
For i = 1, ..., m
    Set $f^{(i)} = \sum_{j=1}^{i-1} L_{i,j} x_j$
    Form $x_i = (I + L_{i,i})^* f^{(i)}$   using Algorithm 2.
```

Hyafil and Kung have shown that for $p = \lceil n^r \rceil$, $1 < r < 3$, and taking

$$m = \left\lceil \frac{68n}{\lceil n^r \rceil^3} \right\rceil ,$$

then

$$T_p(n) \leq \begin{cases} O(n^{2-r} \log n) , & \text{for } 1 < r < \frac{3}{2} \\ O(n^{1-\frac{r}{3}} \log^2 n) , & \text{for } \frac{3}{2} < r < 3. \end{cases}$$

3. Banded Recurrences

In this section, we take the matrix L of (1) to be banded, i.e., of the form

$$L = \begin{bmatrix} L_1 & & & & N & \\ R_1 & L_2 & & & & \\ & R_2 & L_3 & & & \\ & & \cdot & \cdot & & \\ N & & & \cdot & \cdot & \\ & & & & \cdot & \cdot \\ & & & & R_{k-1} & L_k \end{bmatrix} , \tag{12}$$

where L_i is a strictly lower triangular matrix of order m, i = 1,2,...,k, and R_j is an upper triangular matrix of order m, j = 1,2,...,k-1.

3.1 Unlimited Parallelism

We can show that the time and processors required for solving (1) are the same as given by Sameh and Brent [22]. The proof is exactly as that of their theorem 3.1. Before this can be proved, however, we need to establish the following.

Lemma 6:

Let L be a strictly lower triangular 2n by 2n matrix given by

$$L = \begin{bmatrix} L_1 & N \\ R & L_2 \end{bmatrix}, \tag{13}$$

where R is n by n and upper triangular. Let f be a 2n-vector correspondingly partitioned as

$$f^T = (f_1^T, f_2^T) .$$

Thus, the solution of the linear recurrence

$$x = (I_{2n}+L)x + f \tag{14}$$

is given by

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} I_n & N \\ G & I_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \tag{15}$$

where

$$G = (I_n+L_2)^* R ,$$

and

$$y_i = (I_n+L_i)^* f_i , \quad i = 1,2.$$

Proof:

From the structure of L and Theorem 2, we see that

$$x_1 = (I_n+L_1)^* f_1 = y_1 \ .$$

Also, from (13) and (14) we have

$$x_2 = Rx_1 + (I_n+L_2)x_2 + f_2 \ .$$

From Theorem 2, we have

$$x_2 = (I_n+L_2)^* [R_1 x_1 + f_2]$$

$$= Gy_1 + y_2 \ ,$$

proving the lemma. ■

Now we state the main result.

Theorem 3: The linear recurrence

$$x = (I+L)x + f \ ,$$

where L is an n by n strictly lower triangular matrix (12), i.e., $n = km$, is obtained in $(3 + 2 \log m) \log n - (1/2)(\log m)(1 + \log m)$ time steps requiring less than $m(m+1)n/2$ processors.

Proof:

Let $f^T = (f_1^T, f_2^T, \ldots, f_k^T)$. The algorithm of Lemma 6 can be generalized to yield a scheme which is exactly similar to that of Sameh and Brent [22], Theorem 3.1 as follows.

Algorithm 4: Banded recurrences.

Step 1: Set $y_1^{(0)} = (I+L_1)^* f_1$, and

$$[G_{i-1}^{(0)}, y_i^{(0)}] = (I+L_i)^* [R_{i-1}, f_i], \; i=2, 3, \ldots, n/m.$$

Step 2: For $j=1, 2, \ldots, \nu = \log(n/m) - 1$,

Set $r = 2^j m$

$$\text{Set } G_i^{(j+1)} = \begin{bmatrix} N & G_{2i}^{(j)} & \\ N & G_{2i+1}^{(j)} & G_{2i}^{(j)} \end{bmatrix}, \; i=1, 2, \ldots, \frac{n}{2r} - 1, \text{ and}$$

$$y_i^{(j+1)} = \begin{bmatrix} y_{2i-1}^{(j)} \\ G_{2i-1}^{(j)} \, y_{2i-1}^{(j)} + y_{2i}^{(j)} \end{bmatrix}, \; i=1, 2, \ldots, \frac{n}{2r}.$$

Upon termination, $y_1^{(\nu+1)}$ contains the solution vector x. ■

3.2 Limited parallelism

In this section we will assume that $m<p<<n$, where p is the number of processors available. If $p=m$ we can use the column sweep algorithm to solve the recurrence in $2(n-1)$ steps. We will develop a faster algorithm for $p>m$ along the same lines as the algorithm of Theorem 2.3.13 of Kuck and Sameh [18], for solving banded triangular systems of linear algebraic equations. Our results are summarized by the following theorem.

Theorem 4: Given p processors, $m<p<<n$, a linear recurrence with the matrix L of the form (12) can be solved in time

$$T_p = 2(m-1) + \tau \lceil (n-m)/p(p+m-1) \rceil , \qquad (16)$$

where

$$\tau = \max \begin{cases} (2m^2+3m)p - (m/2)(2m^2+3m+5) \\ 2m(m+1)p - 2m \end{cases} . \tag{17}$$

This yields a speedup and efficiency proportional to (p/m) and (1/m) respectively.

<u>Proof</u>: To motivate the approach, consider a recurrence of the form

$$z = \hat{L}\hat{z} + g \text{ of order } s = q + p(p-1)$$

equations with q = mp,

$$\begin{bmatrix} z_1 \\ z_2 \\ \cdot \\ \cdot \\ \cdot \\ z_p \end{bmatrix} = \begin{bmatrix} R_0 & L_1 & & & & \\ & R_1 & L_2 & & & \\ - & - & - & - & - & - \\ & & & & R_{p-1} & L_p \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ \cdot \\ \cdot \\ \cdot \\ z_p \end{bmatrix} + \begin{bmatrix} g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_p \end{bmatrix} \tag{18}$$

Here, L_1 is of order q, $L_i (i>1)$ is of order p, and $R_i (0 \leq i \leq p-1)$ contains non-null elements only on its top m super diagonals, i.e.,

$$R_i = \begin{bmatrix} N & \tilde{R} \\ & N \end{bmatrix}, \tag{19}$$

in which $\tilde{R}$ is upper triangular of order m. The system (18) can be expressed as

$$z_1 = R_0 \; z_0 + L_1 z_1 + g_1,$$

$$z_2 = R_1 \; z_1 + L_2 z_2 + g_2,$$

$$\vdots$$

$$z_p = R_{p-1} z_{p-1} + L_p z_p + g_p .$$

From Theorem 2, we get that the z_j's can be written as

$$z_j = (I+L_j)^*[R_{j-1}z_{j-1} + g_j],\ j=1,\ \ldots,\ p,\quad \text{or}$$

$$z_j = ((I+L_j)^*R_{j-1})z_{j-1} + (I+L_j)^*g_j. \qquad (20)$$

Therefore, we define

$$h_1 = (I+L_1)^*(R_0z_0 + g_1), \qquad (21)$$

$$[G_{i-1},h_i] = (I+L_i)^*[R_{i-1},g_i],\ i=2,\ 3,\ \ldots,\ p. \qquad (22)$$

Using a single processor, equation (21) can be evaluated in $\tau_1=2m^2p$ steps. Using a single processor to evaluate each of the p-1 systems, equation (22) can be evaluated in

$$\tau_2 = [(2m^2 + m)p - (m/2)(2m^2 + 3m - 1)]$$

steps.

Kuck and Sameh [18] have shown that the choice of q=mp keeps the difference in time for evaluating equations (21) and (22) as small as practically possible. Thus the evaluation of equations (21) and (22) can be done in at most $\tau_3 = \max\{\tau_1,\tau_2\}$ steps.

We now see from equations (20), (21), and (22) that z is given by

$$z_1 = h_1$$

$$z_i = h_i + G_{i-1}z_{i-1}, \; i=2, 3, \ldots, p.$$

Since only the last m columns of G_i are non-null, then by using all of the p processors, each z_i can be computed in 2m time steps. Thus, $z_2, z_3, \ldots, z_p$ are all obtained in time

$$\tau_4 = 2m(p-1),$$

and hence the recurrence $z = \hat{L}\hat{z} + g$ can be solved in time

$$\begin{aligned} \tau &= \tau_3 + \tau_4 \\ &= \max \begin{cases} (2m^2+3m)p - (m/2)(2m^2+3m+5), \\ 2m(m+1)p - 2m. \end{cases} \end{aligned}$$

Partition the recurrence $x = (I+L)x + f$ of n equations and bandwidth m+1 into the form

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} I+V_0 & & & \\ U_1 & I+V_1 & & \\ - & - & - & - \\ & & U_k & I+V_k \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{bmatrix} + \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_k \end{bmatrix}$$

where $I+V_0$ is of order m, and $I+V_i$, $i > 0$ (except possibly for $I+V_k$) is of order s, i.e., $k = \lceil (n-m)/s \rceil$, and each U_i is of the form (19). Then, the solution vector x is given by

$$x_0 = (I+V_0)^* f_0 \tag{23}$$

$$x_i = (I+V_i)^* (f_i + U_i x_{i-1}), \; i=1, 2, \ldots, k. \tag{24}$$

Equation (23) can be evaluated in $2(m-1)$ steps using the column sweep algorithm since $p>m$. The k equations in (24) can be solved one at a time using the algorithm developed for solving recurrence (18). Hence using p processors, the recurrence (12) of order n and bandwidth m=1 can be solved in time

$$T_p = 2(m-1) + \tau \lceil (n-m)/p(p+m-1) \rceil,$$

where τ is given by equation (17). ■

4. Linear Systems $x = Ax + b$.

In this section, we will deal with the general, linear system

$$x = Ax + b. \tag{25}$$

In section 1.3, we saw that the solution of the shortest path problem can be obtained by solving the system (25). Due to the form of (25), the n by n matrix A has, in general, its only null elements on the main diagonal. The matrix A is called sparse if it has a small proportion of non-null elements. If we let n_A be the number of non-null elements of A, then we say that A is sparse if $n_A << n^2$.

We reiterate the assumption that the matrix A is definite, that is,the associated graph G has no cycles, the sum of whose arc weights are less than or equal to zero. Now we present two direct methods and two iterative methods for solving the system (25). Since the Matrix A is definite, the iterative methods can also be posed as direct methods. We also discuss the special case when all of the arc weights of the underlying graph G are positive. The section is concluded with a comparison of the methods when the matrix A is sparse.

4.1 Elimination Methods (Carré [2], §7.2, 7.3)

The method of generalized Gaussian elimination was first proposed by Carré [2,3]. We state the method and give time and processor counts for a parallel implementation.

<u>Algorithm 5</u>: Parallel Gaussian Elimination

```
For k=1, 2, ..., n-1
    Set $\alpha_{i,j} = \alpha_{i,k} \times \alpha_{k,j} + \alpha_{i,j}$
        $\beta_i = \alpha_{i,k} \times \beta_k + \beta_i$     } in parallel for (i,j=k+1, k+2, ..., n; i≠j)
```

The resulting upper triangular system can be solved using a method of Section 2. The reduction of the matrix A to upper triangular form can be accomplished in 2(n-1) steps using at most $(n-1)^2$ processors.

The added time and perhaps, added processors necessary to solve the resulting linear recurrence of Gaussian elimination can be avoided, by using the generalized Jordan elimination method of Carré [2, §7.3].

<u>Algorithm 6</u>: Parallel Jordan elimination

```
For k=1, 2, ..., n-1
    Set $\alpha_{i,j} = \alpha_{i,k} \times \alpha_{k,j} + \alpha_{i,j}$
        $\beta_i = \alpha_{i,k} \times \beta_k + \beta_i$
    in parallel for i=1, ..., n; j=k+1, k+2, ..., n; i≠j).
```

Note that upon termination, the vector b contains the solution x. Clearly, Algorithm 6 can be performed in 2(n-1) steps using at most n^2+1 processors. Thus, we see that by using an additional 2n processors over Gaussian elimination, we can solve the system (25) in the same time as it takes to do the elimination step of Gaussian elimination.

4.2 <u>Iterative Methods</u>

In this section we present two iterative methods for solving the system (25). They correspond to the Jacobi and Gauss-Seidel methods for the iterative solution of linear algebraic equations. The methods of Bellman [1] and Moore [21] are sequential versions of the generalized

Jacobi method. Ford and Fulkerson's [10] sequence of scanning the arcs in Ford's method [9] is a sequential version of the generalized Gauss-Seidel method.

The system (25) can be solved using the generalized Jacobi method

$$x^{(k)} = Ax^{(k-1)} + b. \tag{26}$$

Since the graph G associated with the distance matrix A is assumed to be definite, from Theorem 6.1 of Carré [2], for an initial guess of a null $x^{(0)}$, we see that $x^{(n)} = x$.

We observe that the matrix-vector product $Ax^{(k-1)}$ can be formed in parallel in $\log n + 1$ steps using at most n^2 processors. Thus a single iteration requires $\log n + 2$ steps and at most n^2 processors. The overall algorithm can be accomplished in no more than $n(\log n + 2)$ steps using n^2 processors.

In our context, the generalized Jacobi method can be considered as a direct method. Since $x^{(n)}$ is the solution to the system (25), we can write it by repeatedly using equation (26), as

$$x^{(n)} = A^n x^{(0)} + A^{n-1}b + A^{n-2}b + \ldots + Ab + b. \tag{27}$$

Since $x^{(0)}$ was chosen to be null, the equation (27) simplifies to

$$x^{(n)} = A^{n-1}b + A^{n-2}b + \ldots + Ab + b, \tag{28}$$

which can be evaluated in parallel using the following algorithm.

Algorithm 7: Direct Jacobi Method

```
Set $x_0 = b$
For $i=0, 1, \ldots, \nu = \log(\frac{n}{4})$
    $x_{i+1} = (I+A^{2^i})x_i$   (we can also terminate whenever $x_{i+1}=x_i$)
    $A^{2^{i+1}} = A^{2^i}A^{2^i}$
$x = (I+A^{\frac{n}{2}})x_{\nu+1}$   (if necessary)
```

This is the method proposed by Dekel and Sahni [6], for cube connected and perfect shuffle computers. By summing the time counts for Algorithm 7, and observing that the maximum number of processors occurs during the formation of $A^{2^{i+1}}$, we get the following theorem.

Theorem 5: The solution to the system (25) can be obtained in no more than $2\log^2 n + 2\log n - 1$ steps using n^3 processors. ■

We will now discuss a parallel implementation of the generalized Gauss-Seidel method. We begin by observing that the matrix A of the system (25) can be written as

$$A = L + U. \tag{29}$$

Consequently, (25) becomes

$$x = (L+U)x + b, \tag{30}$$

which leads to the generalized Gauss-Seidel iteration

$$x^{(k+1)} = Lx^{(k+1)} + Ux^{(k)} + b.$$

Solving for $x^{(k+1)}$ yields

$$x^{(k+1)} = (I+L)^{*}Ux^{(k)} + (I+L)^{*}b$$

$$\equiv Mx^{(k)} + c. \qquad (31)$$

From formula (2.16) of Sameh and Brent [22], we get that the iteration matrix M and vector c can be formed in $\frac{1}{2}\log^2 n + O(\log n)$ steps using at most $\frac{n^3}{6} + O(n^2)$ processors. Thus, the solution x can be found by the generalized Gauss-Seidel method in n log n + O(n) steps using at most $\frac{n^3}{6} + O(n^2)$ processors.

Carré [2] has shown that the number of iterations required by the generalized Gauss-Seidel method is not greater than the number required by the generalized Jacobi method. It is thus likely that for many problems, the overall work for the generalized Gauss-Seidel method will be less than that of the generalized Jacobi method, when used in an iterative fashion. This potential savings in time has a substantial cost in additional processors.

When used as a direct method, the generalized Gauss-Seidel scheme would use Algorithm 7 applied to the matrix M and vector c of equation (31). Note that an additional $\frac{1}{2}\log^2 n + O(\log n)$ operations are required to form the matrix M and vector c. Hence, for the generalized Gauss-Seidel method to be more efficient than the generalized Jacobi method, Algorithm 7 applied to the matrix M and vector c must take at least 25% fewer iterations than the same algorithm applied to the original matrix A and vector b. Consequently, one would prefer the generalized Jacobi method for solving the system (25).

4.3 Dijkstra's Method

We now consider the special case when all of the arc weights of the associated graph G are positive. In the regular algebra this means that the elements of the matrix A of the system (25) satisfy $\alpha_{i,j} > \varepsilon$ for all i and j. Due to this added assumption, it is easy to see that there can be no linear algebraic analogs for algorithms tailored to these problems.

Dijkstra's method [7] is such an algorithm, iterative in nature with no linear algebraic analog. In Dijkstra's method, nodes of the graph G are separated into two classes, temporary and permanent. All nodes start out temporary and become permanent one at a time. The algorithm terminates once all nodes have become permanent.

Let mode [*] be an array of bits used to indicate whether a given node is temporary or permanent. When mode[i]=0, then node i is temporary. If mode[i]=1, node i is permanent. If we wish to solve the system (25), where $\alpha_{i,j} > \varepsilon$ for all i and j, then Dijkstra's algorithm can be stated as follows:

Algorithm 8: Parallel Dijkstra's Algorithm

The vector b is overwritten by the solution x.

```
Initialize:   mode[i]=0, i=1, 2, ..., n.

          For i=1, 2, ..., n-1
            Find j such that $\beta_j = \sum_{\text{mode}[k]=0} \beta_k$          (32)
            mode[j]=1                                              (33)
            For all k such that mode[k]=0
                $\beta_k = \beta_k + \beta_j \times \alpha_{j,k}$                     (34)
```

We observe that for each iteration, line (32) can be done in $\log(n+1-i)$ steps using $n-i$ processors. Overall, line (32) requires

$$\sum_{i=2}^{n} \log i = n\log\frac{n}{2}$$

operations and at most $n-1$ processors. Similarly line (33) requires one operation each iteration, and line (34) can be performed in parallel at each iteration using two operations and $n-i$ processors. The overall work for a parallel version of Dijkstra's method is $n\log n + 2n - 3$ steps using at most $n-1$ processors. Compared with n iterations of the generalized Jacobi method, Dijkstra's method requires essentially the same time, but far fewer processors ($n-1$ vs. n^2). Thus, for dense matrices A, Dijkstra's method appears to be the iterative method of choice.

4.4 Sparse Matrix Considerations

When the matrix A of the system (25) is large and sparse, that is the number of arcs n_A in the graph G is much less than n^2, direct methods are impractical. Unless the matrix A is banded or has a special structure, undesireable fill-in of the matrix A will occur. Since the economization of storage for large sparse matrices is essential, iterative methods must be used.

Not all iterative methods are appropriate for this problem. The generalized Gauss-Seidel method requires the formation of the iteration matrix $M = (I+L)^* U$. This matrix in general will be dense and hence too costly to store. We will thus restrict ourselves to a discussion of the generalized Jacobi method and Dijkstra's method.

Let q be the maximum number of non-null elements in any row of A. In order to compare the two algorithms, we will use a cost function which is the product of the number of processors and time. For dense matrices, we see that Dijkstra's method has a cost of

$$(n-1)(n \log n + 2n - 3) = n^2 \log n + O(n^2) .$$

Similarly, the generalized Jacobi method has a cost of $n^3 \log n + O(n^3)$ which is about n times greater than that of Dijkstra's method.

For sparse matrices, the generalized Jacobi method requires $n\log q + 2n$ steps and $n_A - n < n(q-1)$ processors. Dijkstra's method still requires $n \log n + 2n - 3$ steps and $n-1$ processors. So, the cost for the generalized Jacobi method becomes

$$n(q-1)(n\log q + 2n) = (q-1)n^2\log q + O(qn^2),$$

while the cost of Dijkstra's method remains the same. When the generalized Jacobi method requires the full n iterations, it is more cost effective than Dijkstra's method if $(q-1)\log q < \log n$. In general, the generalized Jacobi method requires a fraction of n iterations. Let k represent the number of Jacobi iterations required for a given problem. Then if $k(q-1)\log q < n \log n$, the generalized Jacobi method is the superior algorithm. A similar tradeoff between Dijkstra's method and a version of Ford's method was found by Hulme and Wisniewski [14] for sequential algorithms.

References

[1] Bellman, R., "On a routing problem," Quart. Appl. Math., 16, (1958), 87-90.

[2] Carré, B.A., "An algebra for network routing problems," J. Inst. Math. Appl., 7, (1971), 273-294.

[3] Carré, B.A., "An elimination method for minimal-cost network flow problems." In Large sparse sets of linear equations, J.K. Reid (ed.),(Proc. I.M.A. Conf., Oxford, 1970), Academic Press, London, 1971.

[4] Chen, Y.K. and Feng, T., "A parallel algorithm for the maximum flow problem," (summary), Proc. 1973 Sagmore Conf. on Parallel Processing, (1973), 60.

[5] Cruon, R. and Hervé, P., Revue fr. Rech. opér., 34, (1965), 3-19.

[6] Dekel, E., and Sahni, S., "Parallel matrix and graph algorithms," Tech. Report 79-10, Department of Computer Science, University of Minnesota, June 1979.

[7] Dijkstra, E.W., "A note on two problems in connexion with graphs," Numer. Math., 1, (1977), 269-271.

[8] Dreyfus, S.E., "An appraisal of some shortest-path algorithms," Operations Res.,17, (1969), 395-412.

[9] Ford, L.R. Jr., "Network flow theory," P-928, The Rand Corporation, Santa Monica, Ca., August 1956.

[10] Ford, L.R. Jr., and Fulkerson, D.R., Flows in networks, Princeton University Press, Princeton, N.J., (1962), 130-133.

[11] Heller, D., "On the efficient computation of recurrence relations," ICASE, Hampton, Va.; Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pa., 1974.

[12] Hu, T.C., "Revised matrix algorithms for shortest paths," SIAM J., 15, (1967), 207-218.

[13] Hulme, B.L., "MINDPT: a code for minimizing detection probability up to a given time away from a sabatoge target," SAND 77-2039, Sandia Laboratories, Albuquerque, N.M., December 1977.

[14] Hulme, B.L., and Wisniewski, J.A., "A comparison of shortest path algorithms applied to sparse graphs," SAND78-1411, Sandia Laboratories, Albuquerque, N.M., August 1978.

[15] Hyafil, L. and Kung, H.T., "Parallel algorithms for solving triangular linear systems with small parallelism," Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pa., 1974.

[16] Kalaba, R., Proc. 10th Symposium on Applied Mathematics, Am. Math. Soc., (1958), 1-280.

[17] Kuck, D.J., personal communication.

[18] Kuck, D.J., and Sameh, A.H., The Structure of Computers and Computation, Volume II, Wiley, New York, in preparation.

[19] Levitt, K.N. and Kautz, W.H., "Cellular arrays for the solution of graph problems," C. ACM, 15, (1972), 789-801.

[20] Moisil, G.C., Communicările Acad. Republicii Populare Romîne, 10, (1960), 647-652.

[21] Moore, E.F., "The shortest path through a maze," in Proc. Internat. Symposium on Theory of Switching, Part II, The Annals of the Computation Laboratory of Harvard University, Harvard University Press, Cambridge, 30, (1959), 285-292.

[22] Sameh, A.H., and Brent, R.P., "Solving triangular systems on a parallel computer, " SIAM J. Numer. Anal., 14, (1977), 1101-1113.

[23] Varnado, G.B., et al., "Reactor safeguards system assessment and design, Volume I," SAND77-0644, Sandia Laboratories, Albuquerque, N.M., June 1978.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-79-997	2.	3. Recipient's Accession No.
4. Title and Subtitle PARALLEL ALGORITHMS FOR NETWORK ROUTING PROBLEMS AND RECURRENCES			5. Report Date November, 1979
			6.
7. Author(s) John A. Wisniewski and Ahmed H. Sameh			8. Performing Organization Rept. No. UIUCDCS-R-79-997
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US NSF MCS75-21758
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered Technical Report
			14.

15. Supplementary Notes

16. Abstracts

In this paper, we consider the parallel solution of recurrences, and linear systems in the regular algebra of Carré. These problems are equivalent to solving the shortest path problem in graph theory, and they also arise in the analysis of Fortran programs. Our methods for solving linear systems in the regular algebra are analogs of well-known methods for solving systems of linear algebraic equations. A parallel version of Dijkstra's method, which has no linear algebraic analog, is presented. Considerations for choosing an algorithm when the problem is large and sparse are also discussed.

17. Key Words and Document Analysis. 17a. Descriptors

Parallel algorithms, shortest path problem, recurrence, networks, regular algebra, graphs.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 38
	20. Security Class (This Page UNCLASSIFIED	22. Price